MINIMUM ENTROPY RESTORATION USING FPGAS AND HIGH-LEVEL TECHNIQUES

Robin Bruce[1], Caroline Ruet[2], Stephen Marshall[3], Malachy Devlin[4]

[1]Institute for System Level Integration, Livingston, Scotland, EH54 7EG,
phone: +44-117-377-7144, email: rbruce@sli-institute.ac.uk, web:www.sli-institute.ac.uk
[2]Institut National des Sciences Appliquées, Rennes, France, [3]Image Processing Group, Uni. of Strathclyde, Glasgow, Scotland,
[4]Nallatech Ltd., Cumbernauld, Scotland

ABSTRACT

One of the greatest perceived barriers to the widespread use of FPGAs in image processing is the difficulty for application specialists of developing algorithms on reconfigurable hardware. Minimum entropy deconvolution (MED) techniques have been shown to be effective in the restoration of star-field images. This paper reports on an attempt to implement a MED algorithm using simulated annealing, first on a microprocessor, then on an FPGA. The FPGA implementation uses DIME-C, a C-to-gates compiler, coupled with a low-level core library to simplify the design task. Analysis of the C code and output from the DIME-C compiler guided the code optimisation. The paper reports on the design effort that this entailed and the resultant performance improvements.

1. INTRODUCTION

Research into the use of field-programmable gate arrays (FPGAs) in image processing began in earnest at the beginning of the 1990s. Since then, many thousands of publications have pointed to the computational capabilities of FPGAs. During this time, FPGAs have seen the application space to which they are applicable grow in tandem with their logic densities. Reference [1] is a good introduction to FPGA-based reconfigurable computing in general, and [2] describes well the issues surrounding FPGA-based image processing. When investigating a particular application, researchers compare FPGAs with alternative technologies such as Digital Signal Processors (DSPs), Application-Specific Integrated Circuits (ASICs), microprocessors and vector processors. The metrics for comparison depend on the needs of the application, and include such measurements as raw performance, power consumption, unit cost, board footprint, non-recurring engineering cost, design time and design cost. The key metrics for a particular application may also include ratios of these metrics, e.g. power/performance, or performance/unit-cost. The work detailed in this paper compares a 90nm-process commodity microprocessor with a platform based around a 90nm-process FPGA, focussing on design time and raw performance.

The application chosen for implementation was a minimum entropy restoration of star-field images with simulated annealing used to converge towards the globally-optimum solution. The authors did not choose this application in the belief that it would particularly suit one technology over another, but instead selected it as being representative of a computationally intensive image-processing application.

2. MINIMUM ENTROPY DECONVOLUTION

Image restoration using the minimum entropy deconvolution (MED) method is discussed at length in [3]-[8]. However, the algorithm presented in [3] does not offer a precise and clear explanation of the different stages. While the principle is very clearly defined, the different steps of the algorithm are confusing. To implement this algorithm in software and on an FPGA, it is important to understand the complex calculations in order to optimise them. The purpose of this section is then not to redefine the basis of the MED optimisation but to explain with more accuracy the different stages of the simulated annealing algorithm used in this application.

2.1 Simulated Annealing method

Suppose the image distortion system can be modelled as:

$$y(k_1,k_2) = x(k_1,k_2) * h(k_1,k_2) + \xi(k_1,k_2)\,. \qquad (1)$$

What is desired is an estimate of the original image $x(k_1,k_2)$ from the observed image $y(k_1,k_2)$, assuming that the image was distorted by a blurring system whose point spread function (PSF) can be approximated by a Gaussian function:

$$h(d) = \begin{cases} \gamma \exp\left(-\dfrac{m_1^2 + m_2^2}{d}\right), & for \quad m1, m2 = -2,-1,0,1,2 \\ 0, & otherwise \end{cases}. \qquad (2)$$

$$d \in [0,\infty)$$

where m_1, m_2 designates the size of the PSF (m_1, m_2 = -2, -1, 0, 1, 2 for a 5 x 5 filter) and γ is a constant used to normalise the Gaussian function. d corresponds to the width of the PSF and determines the blurring level applied to the image.
$\xi(k_1,k_2)$ is an additive white Gaussian noise defined by a mean and a variance.

The iterative simulated annealing algorithm (SA) consists of starting with an initial estimate for $x(k_1,k_2)$ and searching for changes which minimise the energy function $E(x,h(d))$ defined as:

$$E(x,h(d)) = \frac{\left[\sum_{k_1}\sum_{k_2} x^2(k_1,k_2)\right]^2}{\sum_{k_1}\sum_{k_2} x^4(k_1,k_2)} + \lambda \sum_{k_1}\sum_{k_2}\left[x(k_1,k_2) * h(k_1,k_2) - y(k_1,k_2)\right]^2 \quad (3)$$

The first estimation for x can be chosen to be either the observed image $y(k_1,k_2)$, an empty image or a random image. SA differs from other iterative techniques in that it uses a temperature parameter to avoid getting trapped in a local optimum, something which is generally the case in other methods, such as gradient descent.

In each iteration of the algorithm, a new candidate for the estimate of the original image is computed. Also, a variation in the PSF is produced by varying the parameter blurring level coefficient d. These slight changes result in the system energy $\boldsymbol{E}$ changing by $\Delta\boldsymbol{E}$. If $\Delta\boldsymbol{E}$ is negative the adjustments to the image $x(k_1,k_2)$ and the parameter d are accepted. If $\Delta\boldsymbol{E}$ is positive or zero the adjustments of the image $x(k_1,k_2)$ and the parameter d are accepted with a probability which decreases exponentially with $\Delta\boldsymbol{E}/T$. At the beginning of the process, the temperature T is large and therefore the adjustments are more likely to be accepted, allowing gross features of the image to appear. At low temperature levels, the algorithm is more likely to reject image adjustments, allowing only fine adjustments to the estimated image. When the temperature becomes zero, the procedure stops at the optimal state of minimum energy.

2.2 Algorithm steps

At this point, we use x and y as shorthand for $x(k_1,k_2)$, $y(k_1,k_2)$ respectively to simplify the explanatory text.

- *Step 0:* Set p=0 and initialise α, β, λ, T_p, d_p and x_p.

x_0 can be either the observed image, an empty image or even a random image.

T_0 is set high. The best way to determine a suitable starting temperature is to run the algorithm and note whether or not adjustments are accepted in a good proportion. (100 can be used as a starting point).

d_0 could have a range of [0, ∞] though too large a value would cause the algorithm to fail, the image being too blurred. A range of [0, 20] is therefore more realistic to consider as it gives a PSF that is not too large.

α & λ can be set to 1 to make them neither too small nor too large. However, β has to be very small, 0.0001 being a suitable value.

- *Step 1:* Compute the energy $\boldsymbol{E}_p(x_p, h(d_p))$.

Use (3). Replace x by x_p. Replace $h(d)$ by $h(d_p)$. This means that the Gaussian function has to be determined for each d. Take y as the observed image.

- *Step 2:* Select a candidate solution.

Compute the candidate image x' and the candidate parameter for the Gaussian function d':

$$x'_{p+1} = x_p - \alpha \frac{\partial E}{\partial x_p(n_1,n_2)}. \quad (4)$$

$$d'_{p+1} = d_p - \beta \frac{\partial E}{\partial d_p}. \quad (5)$$

where

$$\frac{\partial E}{\partial x_p(n_1,n_2)} = 4x_p(n_1,n_2)\frac{\sum_{k_1}\sum_{k_2} x_p^2(k_1,k_2)}{\sum_{k_1}\sum_{k_2} x_p^4(k_1,k_2)} \cdot \left[1 - x_p^2(n_1,n_2)\frac{\sum_{k_1}\sum_{k_2} x_p^2(k_1,k_2)}{\sum_{k_1}\sum_{k_2} x_p^4(k_1,k_2)}\right] + 2\lambda \sum_{k_1}\sum_{k_2}\sum_{m_1}\sum_{m_2}\left[\begin{matrix} x_p(k_1-m_1,k_2-m_2) \\ \cdot h_p(m_1,m_2) - y(k_1,k_2)\end{matrix}\right] \cdot h_p(k_1-n_1,k_2-n_2) \quad (6)$$

k_1 & k_2 span the entire image and m_1 & m_2 give the size of the Gaussian function. n_1 & n_2 are used to define a particular pixel of the considered image x_p. The computation of Δx_p is carried out for every pixel of the image to get an overall new value of this image. This value must be computed n^2 times for an image of size n x n.

and:

$$\frac{\partial E}{\partial d_p} = 2\lambda \sum_{k_1}\sum_{k_2}\sum_{m_1}\sum_{m_2}\left[\begin{matrix} x_p(k_1-m_1,k_2-m_2) \\ \cdot h_p(m_1,m_2) - y(k_1,k_2)\end{matrix}\right] \cdot \sum_{m_1}\sum_{m_2} \frac{(m_1^2+m_2^2)}{d_p^2} x_p(k_1-m_1,k_2-m_2) \cdot h_p(m_1,m_2) \quad (7)$$

k_1, k_2, m_1 and m_2 have the same definition as before. This computation is unique for each iteration of the algorithm.

- *Step 3:* Compute the energy $\boldsymbol{E}'_{p+1}(x'_{p+1}, h(d'_{p+1}))$. Let:

$$\Delta E = E'_{p+1} - E_p. \quad (8)$$

- *Step 4:* If:

$$\exp\left(-\frac{\Delta E}{T_p}\right) > r. \quad (9)$$

Then:

$$x_{p+1} = x'_{p+1}, \quad d_{p+1} = d'_{p+1} \quad and \quad T_{p+1} = T_p. \quad (11)$$

where r is a random number on the interval [0,1].

Else:

$$x_{p+1} = x_p, \quad d_{p+1} = d_p \quad and \quad T_{p+1} = f(T_p). \quad (12)$$

where f(.) is a decreasing function. The simplest function would be: $f(x) = x - 1$.

- *Step 5:* $p = p + 1$.

If the temperature T is not zero, the termination condition is not satisfied, go to *Step 1*.

- *Step 6:* Output x_{p+1} is the estimation image.

The last x_{p+1} image is the estimate of the original image.

The data used in calculations in the algorithm begin as 8-bit integer pixel values. The 24 bits of precision of IEEE754 single precision suffices for all the calculations in the algorithm. Floating-point calculations are required because there are parts of the algorithm that require higher dynamic range than provided by integer arithmetic. The dynamic range requirements are fully satisfied by single precision, so there is no requirement for double-precision calculations. Nallatech's DIME-C compiler, a C-to-Gates compiler that targets FPGA computing boards was used. The compiler is well-suited to floating-point computation, uses a subset of ANSI C and allows for the inclusion of libraries of low-level functional cores.

3. HIGH-LEVEL TECHNIQUES FOR FPGAS

3.1 Traditional FPGA Programming

Field-Programmable Gate Arrays are programmed by means of a *bitstream* or *bitfile* that tells the chip how to configure its internal logic, memory and routing resources. Without this bitstream the chip has no functionality at all. The "traditional" method of obtaining an FPGA bitstream is to describe the desired system in terms of synchronous electronic components using a hardware description language (HDL). The most well-known of these HDLs are VHDL and Verilog. To go from a functional specification to a functioning bitstream involves writing HDL, then simulating it. Due to the low level description and verbose nature of these HDL languages this can be a lengthy and error-prone process using expensive tools. Once the HDL is functioning correctly in simulation, it is passed through synthesis tools. For high-performance computing applications, this "traditional" process may result in good performance and low resource use, but it requires an expertise that most application developers do not possess and requires an investment of time that would be unreasonable for most applications.

3.2 High-Level FPGA Programming

There has been a concerted research effort aimed at developing design techniques for reconfigurable computing that better suit application-domain specialists [9]-[11]. Among the high-level tools that promise to simplify the task of implementing an algorithm is DIME-C. DIME-C is a compiler that turns high-level code into a combination of VHDL and pre-synthesized logical netlists. The C that DIME-C can compile is a subset of ANSI C. This means that while not everything that can be compiled using a standard C compiler can be compiled by DIME-C, all source code that can be compiled in DIME-C can also be compiled using a standard C compiler. This allows for rapid functional verification of algorithm code before compilation to FPGA hardware.

Application developers write code as standard ANSI C, avoiding certain constructs such as pointers. The compiler aims to extract obvious parallelism within loop bodies as well as to pipeline loops wherever possible. In nested loops, the compiler pipelines the innermost loop. One must also ensure loops do not break any of the rules for pipelining. The code must be non-recursive, and must not access memory arrays more times per cycle than can be accommodated by the underlying memory structure. Beyond these considerations the user does not need any knowledge of hardware design in order to produce VHDL code of pipelined architectures that implement algorithms. DIME-C supports bit-level, integer and floating-point arithmetic. The compiler also supports the inclusion of support libraries that allow users to implement functions previously created either in DIME-C or via a more traditional design process directly using HDLs. Additionally, the compiler seeks to exploit the essentially serial nature of the programs to resource share between sections of the code that do not execute concurrently. This means the compiler can implement complex algorithms that demand many floating-point operations, provided no concurrently executing code aims to use more resources than are available on the device. Such a resource-sharing optimisation would be extremely difficult to implement manually using HDL. The compiler displays its temporal scheduling visually and produces a report file that together show the parallelisation of the user code. Figure 1 below shows the programming process used. DIMETalk is used here is equivalent to a software linker to link the DIME-C code to the necessary memory structure and the specific hardware platform.

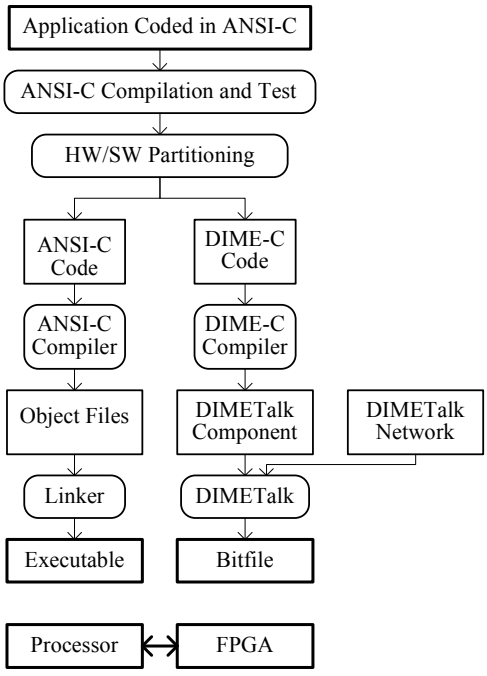

Figure 1 – Programming process for reconfigurable computing using Nallatech tools and hardware

3.3 Core Libraries

DIME-C allows for the inclusion of core libraries. Core libraries allow users to use FPGA functions that have been developed, tested and packaged. Reference [12] gives detail on the motivations for integrating core libraries into high-level tools. The reference also discusses the creation of a math library that is used in the context of this project. This math library has been created using standard HDL techniques, then packaged in such a way as to be indistinguishable from the math library used in ANSI C to the tools user. The *exponential* function and a *pseudo-random number generator* function are used in the simulated annealing algorithm that is implemented here. These functions were created in VHDL, then packaged into a core library to allow them to be called in an identical manner to their ANSI C counterparts. Without access to this math library, the implementation of the simulated annealing algorithm would have represented a far more significant research challenge. This paper is, believed to be, the first publication on the use of this mathematical core library in an actual application.

The difference between the use of library-enabled high-level FPGA languages such as DIME-C and a traditional HDL approach is marked. It is analogous to the difference between programming in the C language with access to function libraries and creating programs in assembler without any support libraries.

4. IMPLEMENTATION

Two of the authors involved in the work took the role of *application specialists*, in that they focussed primarily on developing the image processing application and obtaining a working software implementation. The other two authors were *reconfigurable-computing* specialists, who looked to interact with the application specialists to help them make the transition to a hardware implementation, instructing them on how to get the most from the design tools used.

4.1 Software and Hardware Platforms

The software platform was a 3.2 GHz Intel Pentium D processor with 2 GB of DRAM, with the GNU C compiler, version 3.4.2. The targeted FPGA platform was a Nallatech H101-PCIXM card, with the DIME-C compiler. The H101-PCIXM card has a Xilinx Virtex-4 LX100 FPGA, 512 MB of DRAM and 4 banks of 200MHz, 4 MB SRAM.

4.2 ANSI C Software Implementation

In the first stage of implementation, the application specialists developed the algorithm from theory to implementation in ANSI C on a commodity microprocessor. This stage represented the majority of the time spent in the project, around 100 person-hours. The application specialists carried out this work without any significant input from the reconfigurable-computing specialists. Once the algorithm was functioning satisfactorily in software, the second stage began.

4.3 DIME-C Hardware Implementation

In the second stage of the algorithm, the application specialists and the reconfigurable computing specialists worked together to optimise the algorithm, to migrate it to the DIME-C environment, and to characterise its performance. This process took approximately 25 person-hours. The ANSI C implementation was optimised for performance in order to make for the fairest comparison. The software performance was improved by several orders of magnitude during this time; otherwise, the software-hardware comparison would have been more weighted in favour of the FPGA.

A typical procedure for the porting of algorithms to software is to first profile the software-implemented algorithm, then implement on the FPGA the functions that represent the majority of the calculation time. There are disadvantages to this approach. It neglects to take into account the data transfers that are necessary between the reconfigurable computing platform and the microprocessor-based system before and after each function call. When factored in, these data transfers can negate any performance improvement in the hardware-implemented function. The approach taken here was to implement the entire algorithm on the FPGA, so that the data transfer time is negligible in comparison to the compute time. This means that all improvements to the computation time of a section of the algorithm translate into a measured performance improvement.

4.3.1 Implementation Process

The implementation process consisted of the following steps:

1. Create a DIME-C project using the original source from the ANSI C project.
2. Adapt the source to allow compilation in both DIME-C and ANSI C environments.
3. Take advantage of the most obvious pipelining opportunities to create 1st FPGA implementation.
4. Examine the source code and the output of DIME-C, create an equation that expressed the runtime of the algorithm in cycles, as a function of the parameters of the algorithm, divided into key sections.
5. Determine for a typical set of algorithm parameters the section that takes up the majority of the runtime, and optimise the DIME-C for this section to create the 2nd FPGA implementation.
6. Repeat sections 4&5 to produce the 3rd & 4th FPGA implementations.

4.3.2 Algorithm Performance Characterisation

When an algorithm is compiled in DIME-C the user is presented with a graphical representation of the hardware that has been generated for implementation on the FPGA. This graphical representation informs the user which sections of the code have been pipelined and parallelised, and the latency of operations, function calls and loops. Using this information in conjunction with the source code for the algorithm, one can derive a characteristic function for the algorithm. This gives the number of cycles to run the algorithm as a function of the algorithm's parameters and of the latencies

of the various loops and sections of the code. It is possible to significantly simplify the resultant characteristic expression by factoring out the sections that do not appreciably contribute to the total run time.

The characteristic expression derived for the FPGA-implemented program was as follows:

$$N_{cycles} = (De\_Dx(n_1, n_2, c, l) + Filter(n_1, n_2, c, l) + Other(n_1, n_2, c, l)) \cdot n\_iter \quad . \quad (13)$$

Where *De_Dx* corresponds to equation (6), *Filter* is the application of the Gaussian filter to the image and *Other* is all other operations in the algorithm. n_iter is the number of iterations taken to carry out the simulated annealing algorithm. n_1 and n_2 are the dimensions of the PSF, c and l are the column and line widths of the image respectively.

As the *Filter* section was improved, the performance of the algorithm improved. The evolution of the characteristic equation for *Filter* through three incarnations of the DIME-C source can be seen below:

$$\begin{aligned} Filter_{FPGA\_1}(n_1, n_2, c, l) &= 3 \cdot (2n_2 + 101) \cdot c \cdot l \cdot (2n_1 + 1) \\ Filter_{FPGA\_2}(n_1, n_2, c, l) &= (\frac{c \cdot l}{2}) \cdot (\frac{3n_1 n_2}{2} + 151) \\ Filter_{FPGA\_3}(n_1, n_2, c, l) &= 2 \cdot (c \cdot l + 111) \end{aligned} \quad . \quad (14)$$

The 4th FPGA implementation improved the performance of De_Dx. De_Dx would remain the focus of a 5th implementation. Table 1 below shows how the four successive implementations of the algorithm on the FPGA platform compared with the optimised microprocessor implementation. Data transfer times were negligible and did not contribute to the result. The results below are for an image of size 800×600, with a 5×5 PSF. The algorithm took 100 iterations to complete and the FPGA clock rate was 100MHz.

5. CONCLUSIONS

Latest generation reconfigurable computing platforms are suitable for the implementation of entire image processing algorithms that require significant levels of floating point computation. When using high-level languages significant speedup can be measured. Core libraries further simplify the task of implementing algorithms. The design time that was required to port the algorithm was not disproportionate in comparison to the time spent developing and implementing the algorithm. Developing characteristic expressions for the different algorithmic sections aided in identifying the parts of the algorithm that would most benefit from optimisation, hence speeding up the development process.

REFERENCES

[1] J. Villasenor and B. Hutchings, "The flexibility of configurable computing". *IEEE Signal Processing Magazine*, 15, 5, pp. 67-84, 1998.

[2] C. T. Johnston, K. T. Gribbon and D. G. Bailey, "Implementing Image Processing Algorithms on FPGAs. *Proceedings of the Eleventh Electronics New Zealand Conference, ENZCon'04*, 118-123, 2004

[3] Wu H, Barba J, "Minimum entropy restoration of star field images". *Systems, Man and Cybernetics, Part B, IEEE Transactions on*, Vol. 28, No. 2., pp. 227-231., 1998.

[4] R.Wiggins, "Minimum entropy deconvolution," *Geoexplor*, vol. 16, pp. 21–35, Jan. 1978.

[5] D. Donoho, "On minimum entropy deconvolution," in *Applied Time Analysis, II.* New York: Academic, 1981.

[6] R. Wiggins, "Entropy guided deconvolution," *Geophys.,* vol. 50, pp. 2720–2726, Dec. 1985.

[7] M. Ooe, and T. J. Ulrych, "Minimum entropy deconvolution with an exponential transformation," *Geophys. Prosp.,* vol. 27, pp. 458–473, 1979.

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimisation by simulated annealing," *Science,* vol. 220, pp. 671–680, May 1983.

[9] I. Alston and B. Madahar, "From C to netlists: hardware engineering for software engineers?" *Electronics & Communication Engineering Journal*, 14, 4, pp. 165-173, 2002.

[10] R. Rinker, J. Hammes, W. A. Najjar, W. Bohm and B. Draper. "Compiling image processing applications to reconfigurable hardware." *Proceedings. IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, *Boston, MA, USA,* pp. 56-65, 2000

[11] J. Hammes, B. Rinker, W. Bohm, W. Najjar, B. Draper and R. Beveridge. "Cameron: high level language compilation for reconfigurable systems." *Proceedings International Conference on Parallel Architectures and Compilation Techniques*, *Newport Beach, CA, USA,* pp. 236-244, 1999

[12] R. Bruce, M. Devlin, S. Marshall, "Lessons Learned Implementing the VSIPL API on Reconfigurable Computers." *Proceedings of Military and Programmable Logic Devices International Conference, Washington DC, USA,* Submission 229, 2006.

	Software	1st FPGA	2nd FPGA	3rd FPGA	4th FPGA
Cycles		7.98×10^{10}	8.72×10^{10}	4.30×10^{10}	2.59×10^{10}
Time in Seconds	216	798.00	87.24	42.96	25.92
Speedup vs. Software	1	0.27	2.48	5.03	8.33
% Contribution of:					
De_Dx		5.02	45.94	93.29	88.91
Filter		94.74	51.86	2.24	3.71
Rest of Algorithm		0.24	2.20	4.47	7.38

Table 1 – Performance Comparison of FPGA implementations versus software